

# The classes P and NP

## Reading

Section 7.2

Practice problems (page 294):

## The class P

Computational problems are considered “tractable”, if their running time is polynomial. While a polynomial can grow with  $n$ , it does so in much more reasonable ways than an exponential.

The class  $P$  consists of all languages that are decidable in polynomial time by a deterministic single-tape Turing Machine.

A number of well-known problems belong to the class  $P$ , and looking back at algorithms you have learned in your other classes you can find more examples.

We will now consider several popular members of the class  $P$

## The Path problem

The path problem is represented by the language:

$$\text{PATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph and has a directed path from } s \text{ to } t \}$$

$$\text{PATH} \in P$$

One relevant question is how we represent the graph  $G$ . There are various ways, and they all involve space polynomial in the number of nodes  $n$ . Since the class  $P$  is effectively invariant under such transformations, we can consider  $n$  to be the “size” of our problem.

A “naive” approach to solving this problem would attempt to consider all possible “paths”, which are  $m^m$  if we denote by  $m$  the number of edges. This would not be polynomial ( $m$  is essentially  $O(n^2)$ ).

But of course there are more efficient ways, essentially involving marking of the vertices:

1. Start by marking the vertex  $s$ .

2. Repeat until nothing new is marked:

- Go through the edge list.
- If the source is marked and the target is not, mark the target.

3. See if  $t$  is marked.

Clearly the time-intensive portion is the second part. It will have to run once for each vertex (because each time it must mark a new vertex or we are done), and it takes time  $O(m)$  to run through the edge list. So its total running time is  $O(n^3)$ , polynomial.

### Relatively prime numbers

Another popular polynomial-time problem is the determination of whether two numbers are relatively prime or not. An important consideration here is the size of the input.

A number  $N$  is stored in base 2 using  $n = O(\log N)$  space, by simply using the base-2 representation of  $N$ .

Now consider the problem:

$$\text{RELPRIME} = \{ \langle x, y \rangle \mid x, y \text{ are relatively prime} \}$$

$$\text{RELPRIME} \in P$$

The size of the input is here  $O(\log N)$  where  $N$  is the largest of the two numbers. This is important to keep in mind. For instance a naive approach would be to go through each number  $d$  up to  $x, y$  and divide into them to see if it is a common factor. But this would take too long: There are in general  $O(N)$  such numbers, and  $N = 2^{O(n)}$  is exponential in the size of the input. Essentially, it would take too long.

Instead we will perform the well-known Euclidean division algorithm:

$E =$  On input  $\langle x, y \rangle$ :

A. Repeat until  $y = 0$ : 1. Compute  $x = x \bmod y$  2. Swap  $x$  and  $y$  B. The resulting  $x$  is the greatest common divisor of  $x, y$ . If it is 1 then we accept, otherwise we reject.

The key intuition here is that each repetition is effectively cutting the size of the inputs  $x, y$  by at least a half every second time through the loop. So the number of A steps needed is  $O(\log N)$ . Each of those steps is also polynomial in  $\log N$ , the length of the representations of  $x, y$ .

## Context Free Languages are in P

If  $L$  is a context free language, then it belongs to the complexity class  $P$ .

We will only outline the proof here, and refer to the book for details. We start by considering a CFG in Chomsky Normal Form for the language. Then we know that if we want to derive a string of length  $n$ , we will require exactly  $2n - 1$  steps in our derivation.

One naive approach therefore would be to try out all derivations of  $2n - 1$  steps, but this turns out to not be polynomial in  $n$ .

The solution involves the idea of **dynamic programming**, whereupon we store the results of “smaller” problems to avoid having to repeat them. The idea goes as follows:

1. Given a target word  $w = w_1w_2 \cdots w_n$ .
2. We will progressively fill up an  $n \times n$  table, whose  $(i, j)$  entry contains the totality of variables that can derive the substring  $w_iw_{i+1} \cdots w_j$ . Only one half of the table will need to be filled.
3. The diagonal corresponds to the individual substrings  $w_i$  consisting of one character. We can fill those in by a quick scan of the production rules to find any productions  $A \rightarrow w_i$ .
4. For other  $(i, j)$  entries: Consider all splits of the substring  $w_iw_{i+1} \cdots w_j$ , and for each split consider for each rule  $A \rightarrow BC$  whether  $B$  can produce the first part of the split and  $C$  can produce the second part. Then  $A$  can produce the substring.
5. Repeat this with  $i, j$  pairs progressively further from each other (so we fill from the main diagonal and going outwards, one diagonal at a time).
6.  $w$  is in  $L$  if and only if the start variable  $S$  is in the  $(1, n)$ -th entry.

To examine the complexity of this process, the dominant step is 4. There are in the order of  $O(n^2)$  times that step 4 will need to be repeated. It also takes  $O(n)$  time to do step 4 (the number of non-terminals in the grammar is constant, and step 4 requires at most that number times  $n$  steps). A total running time of  $O(n^3)$ .

## The class NP

The class  $NP$  consists of problems that are “verifiable in polynomial time”. What this means is that it might not be possible to determine in polynomial time, given an instance, whether it is

TODO